Autonomous Computer Vision-Based Navigation Robot

Karan Humpal
Electrical and Computer Engineering
PID: A69036762

Figure 1: Autonomous robot platform with Pi Camera and 2WD base

Abstract

This project consists of a fully 3d printed, autonomous robot that navigates indoor environments using only visual input from a single camera. The system is built using a Raspberry Pi 4, Pi Camera Module, and an Arduino Uno controlling motors through an L298N H-Bridge. A convolutional neural network (CNN) was trained on a custom-labeled dataset to classify scenes into four navigational categories: clear path, obstacle left, obstacle right, and stop. Real-time inference is performed on the Raspberry Pi, which then sends movement commands to the Arduino via serial communication. The robot successfully demonstrates smooth and intelligent movement without any range sensors, relying entirely on deep learning-based perception. The final trained model achieved 91% classification accuracy, and the robot was tested in a structured indoor course with real-world obstacles.

1 Introduction

The problem addressed in this project is how to enable autonomous obstacle avoidance using only a camera. Most autonomous vehicles rely on a combination of sensors such as LIDAR, ultrasonic rangefinders, and infrared arrays to detect obstacles and make navigation decisions. However, these sensors increase system complexity and cost. The goal of this project is to align with the curriculum taught in ECE285 and to see if we can create a fully autonomous robot that only relies on a single camera and trained CNN. The hypothesis is that a deep learning model can learn to identify directional cues (e.g., whether the path is clear, or if there is an obstacle on the left or right) from labeled images captured by the robot's onboard camera.

The proposed solution involves using a Raspberry Pi 4 equipped with a wide-angle Pi Camera Module to capture live images of the environment. These images are processed in real time using a CNN trained to classify the scene into one of four categories: `clear`, `left`, `right`, or `stop`. The classification result is sent over a serial connection to an Arduino Mega, which executes appropriate motor commands using an L298N H-Bridge. The Arduino controls motor speed and direction to achieve forward movement, gradual turning, or a reverse-and-spin maneuver when a full stop is required. This is done through a rather simple state machine written in the Arduino IDE.

After training the model on a custom dataset of approximately 250 labeled images and tuning the control logic, the robot achieved a classification accuracy of 91%. In real-world tests, it successfully navigated an obstacle course made from everyday objects such as chairs and walls, demonstrating that visual-only navigation is viable and effective in controlled indoor environments.

2 Related Work

Vision based navigation has been used a lot in academic and commercial robotic systems. One foundational work is NVIDIA's DAVE-2 system [1], which used a CNN to directly map camera images to steering commands in a self driving car. Although the scale and domain differ a lot, this work demonstrates that raw camera input can be sufficient for real-time navigation if processed with the right neural network architecture.

Another related approach is the use of lightweight CNNs such as MobileNet [2], which are designed for deployment on edge devices with limited compute power. While MobileNet was not used directly in this project, the idea of using compact and efficient networks is reflected in the custom CNN architecture designed for the Raspberry Pi.

Platforms like PiCar-V and OpenBot have demonstrated low cost robot autonomy using computer vision. These platforms often combine classical image processing techniques (e.g., color segmentation, optical flow) with handcrafted rules. In contrast, this project relies entirely on supervised deep learning for perception and decision-making, eliminating the need for manual tuning of vision pipelines.

References

[1] Bojarski, Mariusz, et al. "End to end learning for self-driving cars." *arXiv preprint arXiv:1604.07316* (2016).

[2] Howard, Andrew G., et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

[3] OpenBot Project. `https://github.com/intel-isl/OpenBot`

3 Method

The robot system was developed across three domains: mechanical design, electrical integration, and embedded inference and control. This section describes the design process and implementation details in each of these areas.

3.1 Mechanical Design (CAD and Fabrication)

The physical chassis of the robot was fully designed in SolidWorks and fabricated using my own Bambu Lab Carbon X1 3D printer. The robot features a compact two-level structure with a central 2-wheel drive system and four static skids for stabilization. The two DC brushed motors are positioned at the center of the lower platform, providing balanced propulsion and turning capabilities.

The design process involved multiple iterations and trial and error in SolidWorks to optimize motor placement, battery fit, and weight distribution. The lower deck contains two motor mounts and slotted compartments for securely housing both the 12V Li-ion battery and an auxiliary 9V battery during early testing. Holes were designed for motor shaft clearance and cable routing. The Arduino Uno was placed on the lower level to simplify motor and power wiring.

The upper deck was press-fit into the lower level and included mounting holes specifically for the Raspberry Pi Camera Module. This upper level serves as the electronics hub, holding the Raspberry Pi 4 and its 20,000mAh 65W USB-C power bank. The Pi Camera was mounted facing forward at a consistent 4-inch height above ground. A cable channel allowed a USB connection from the Raspberry Pi to the Arduino to pass between levels.

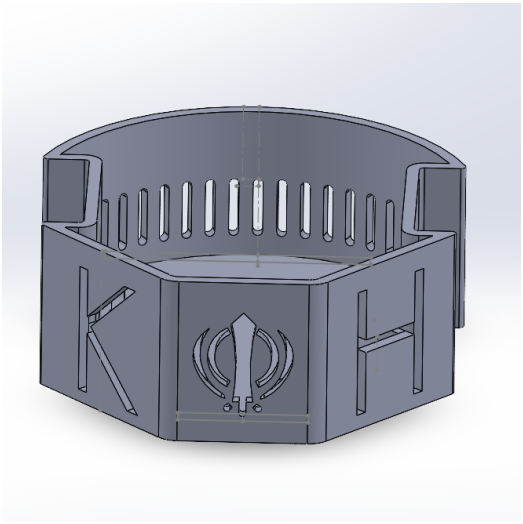


Figure 2: Bottom Level CAD

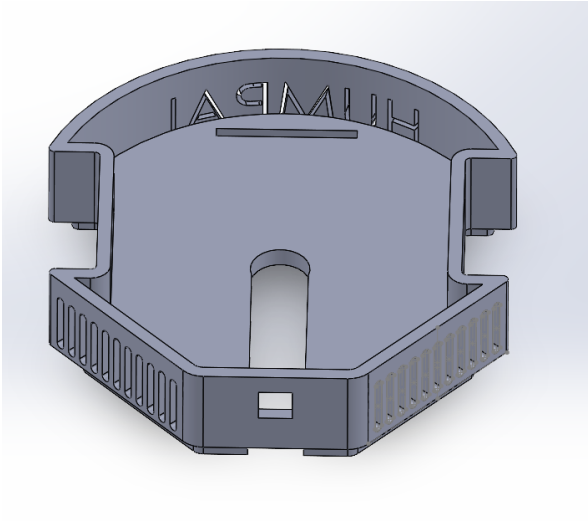


Figure 3: Top Level CAD

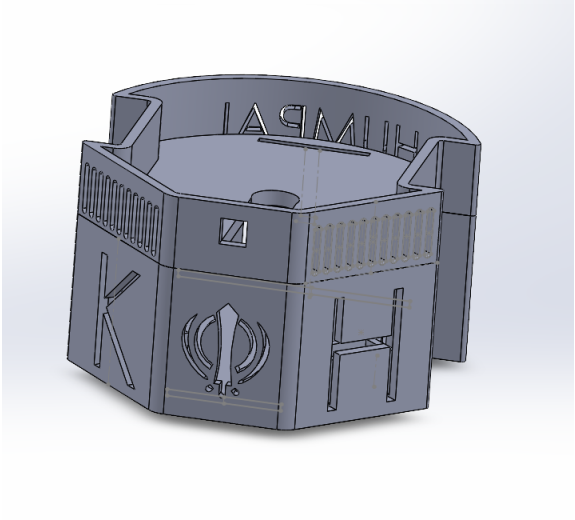


Figure 4: Full Mate

3.2 Electrical System

The robot uses two 12V brushed DC motors, each soldered with designated positive and negative leads. These leads were connected to the output terminals of an L298N dual H-Bridge motor driver. The motors were then mounted and securely screwed into the lower deck of the chassis. Early prototypes powered the motors using 9V D batteries, which proved insufficient due to current limitations and were replaced by a 12V 2400mAh Li-ion battery. This battery was connected to the L298N's 12V input, and its ground was shared with both the Arduino GND and the motor driver's GND to ensure a common reference.

PWM-enabled pins from the Arduino were connected to the ENA and ENB pins of the L298N to control motor speed, while IN1–IN4 were wired to digital pins for direction control. The Arduino itself was powered directly through USB from the Raspberry Pi, which also served as the serial communication link.

The Raspberry Pi Camera Module was connected to the Pi via the CSI ribbon interface, and the Pi was powered using a dedicated 20,000mAh 65W USB-C power bank. All components were properly grounded, and power delivery was isolated between motor and logic systems for noise reduction.

Figure 5: Bottom Level wiring, Before mounting of Arduino and Battery

Figure 6: Top Level, Raspberry Pi and Power Bank

3.3 Software Architecture and Inference

The robot's autonomy is driven by a lightweight PyTorch convolutional neural network (CNN) deployed directly on the Raspberry Pi. Instead of using OpenCV for frame capture, the system uses `libcamera-jpeg` to capture static frames every second, which are then resized to 128x128 and passed through the CNN for classification.

The CNN architecture consists of two convolutional layers with ReLU activations and max-pooling, followed by a fully connected layer that outputs one of four navigational classes: `clear`, `left`, `right`, or `stop`. The model was trained on JupyterHub using GPU acceleration and reached 91% validation accuracy.

During runtime, the Pi captures a frame, performs classification, and sends the result over a serial USB connection to the Arduino. Each frame takes approximately one second to process, enabling near real-time inference without overloading the Pi CPU.

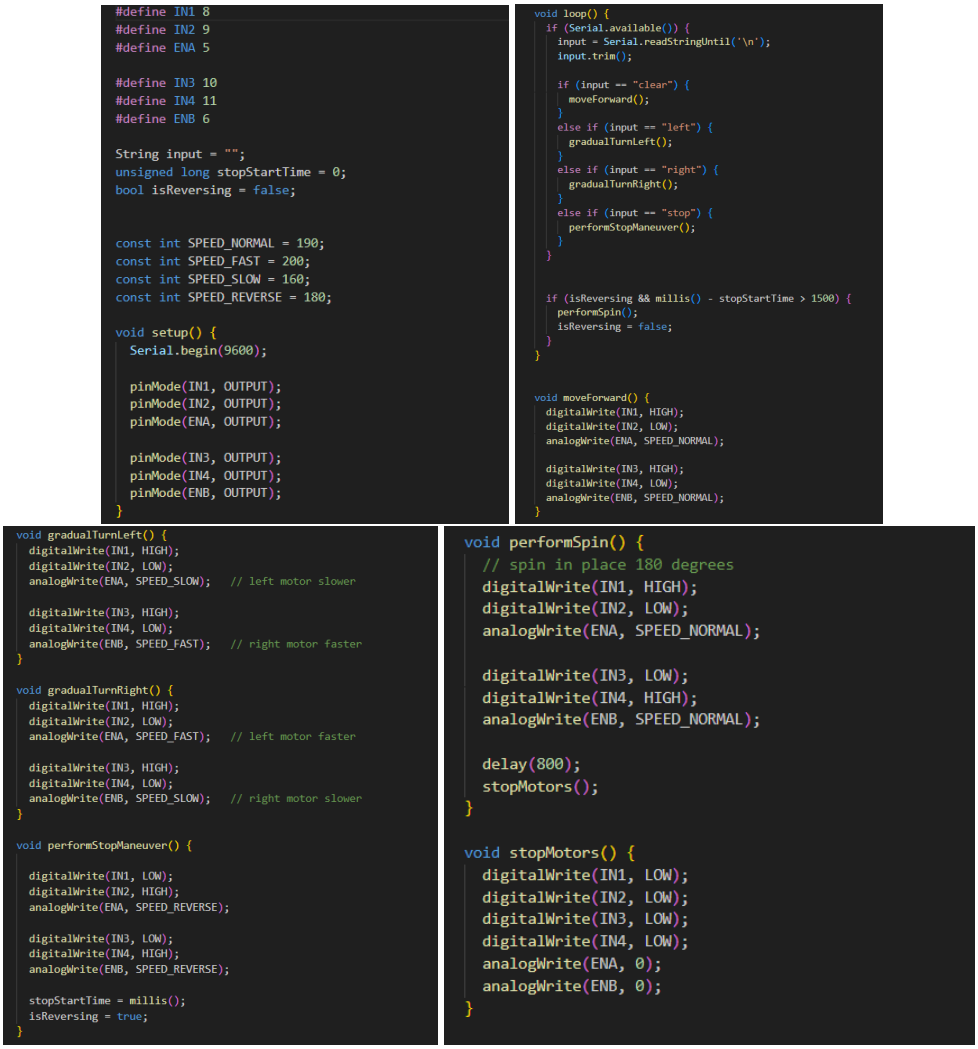
```python
class FinalCNN(nn.Module):
    def __init__(self, num_classes=4):
        super(FinalCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        dummy_input = torch.zeros(1, 3, 128, 128)
        x = self.pool(F.relu(self.conv1(dummy_input)))
        x = self.pool(F.relu(self.conv2(x)))
        self.flattened_size = x.view(-1).shape[0]

        self.fc1 = nn.Linear(self.flattened_size, 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)


model = FinalCNN(num_classes=4)
```

Figure 7: Convolutional Neural Network

3.4 Motor Control and Navigation Logic

The Arduino implements a state machine that receives commands via serial and translates them into motor control actions. These include:

- **clear**: Both motors run forward at equal speed
- **left/right**: Both wheels move forward, but one is slower to create a smooth, gradual turn
- **stop**: The robot briefly reverses, then performs a timed 180-degree spin in place to reorient



```
#define IN1 8
#define IN2 9
#define ENA 5

#define IN3 10
#define IN4 11
#define ENB 6

String input = "";
unsigned long stopStartTime = 0;
bool isReversing = false;


const int SPEED_NORMAL = 190;
const int SPEED_FAST = 200;
const int SPEED_SLOW = 160;
const int SPEED_REVERSE = 180;

void setup() {
  Serial.begin(9600);

  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENA, OUTPUT);

  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(ENB, OUTPUT);
}
```

```
void loop() {
  if (Serial.available()) {
    input = Serial.readStringUntil('\n');
    input.trim();

    if (input == "clear") {
      moveForward();
    }
    else if (input == "left") {
      gradualTurnLeft();
    }
    else if (input == "right") {
      gradualTurnRight();
    }
    else if (input == "stop") {
      performStopManeuver();
    }
  }


  if (isReversing && millis() - stopStartTime > 1500) {
    performSpin();
    isReversing = false;
  }
}


void moveForward() {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, SPEED_NORMAL);

  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, SPEED_NORMAL);
}
```

```
void gradualTurnLeft() {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, SPEED_SLOW);   // left motor slower

  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, SPEED_FAST);   // right motor faster
}

void gradualTurnRight() {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, SPEED_FAST);   // left motor faster

  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, SPEED_SLOW);   // right motor slower
}

void performStopManeuver() {

  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  analogWrite(ENA, SPEED_REVERSE);

  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  analogWrite(ENB, SPEED_REVERSE);

  stopStartTime = millis();
  isReversing = true;
}
```

```
void performSpin() {
  // spin in place 180 degrees
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, SPEED_NORMAL);

  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  analogWrite(ENB, SPEED_NORMAL);

  delay(800);
  stopMotors();
}

void stopMotors() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);
  analogWrite(ENA, 0);
  analogWrite(ENB, 0);
}
```

Figure 8: Full Arduino Code for Motor Control

PWM speed levels were tuned to account for the weight and traction of the bot. Speeds were carefully selected to prevent jerky starts or stalling. Turning behavior was improved by keeping both motors active during turns rather than pivoting around one wheel, resulting in smoother arcs.

This separation between visual inference and actuation allowed each subsystem to remain simple and maintainable, while still enabling robust navigation behavior.

4 Experiments

4.1 Dataset Collection and Format

The dataset was created by manually driving the robot through an indoor obstacle course composed of common household objects such as chairs, walls, shoes, backpacks, and even my own legs. To ensure a controlled and consistent visual environment, I cleared out a small section of a room and placed three to four obstacles at various positions in each run. The Pi Camera, mounted at a fixed height of approximately 4 inches, captured the robot's forward-facing perspective using `libcamera-jpeg`.

I used a Python script running on the Raspberry Pi to capture and label each frame. The script waited for key presses to assign class labels: `w` for `clear`, `a` for `left`, `d` for `right`, and `s` for `stop`. A live feed preview was opened simultaneously using `libcamera-vid`, which allowed me to see what the robot's camera was seeing in real time. I controlled the labeling process entirely through a VNC

connection to the Pi from my laptop, making the workflow efficient and portable. This method also ensured consistent framing and lighting during data collection.

Each image was saved in JPEG format and resized to 128x128 pixels. The dataset was organized into four class folders corresponding to each navigation decision. Initially, the dataset was imbalanced, with fewer samples for the `left` and `stop` classes. After identifying this issue, I purposefully collected additional examples for those cases to ensure near-balanced class representation. The final dataset consisted of approximately 250 images, with 50–70 samples per class.

```python
import cv2
import os
from datetime import datetime

LABEL_KEYS = {
    ord('w'): 'clear',
    ord('a'): 'left',
    ord('d'): 'right',
    ord('s'): 'stop'
}

SAVE_PATH = 'dataset'
RES = (128, 128)

cap = cv2.VideoCapture(0)

print("Camera ready. Press W/A/S/D to save a labeled image. Press Q to quit.")

while True:
    ret, frame = cap.read()
    if not ret:
        print("Camera read failed")
        break

    resized = cv2.resize(frame, RES)
    cv2.imshow("Live Feed", resized)

    key = cv2.waitKey(1)
    if key == ord('q'):
        break
    elif key in LABEL_KEYS:
        label = LABEL_KEYS[key]
        folder = os.path.join(SAVE_PATH, label)
        os.makedirs(folder, exist_ok=True)
        filename = datetime.now().strftime("%Y%m%d_%H%M%S%f") + ".jpg"
        cv2.imwrite(os.path.join(folder, filename), resized)
        print(f"[{label}] image saved")

cap.release()
cv2.destroyAllWindows()
```

Figure 9: Script for Data collection, run on the PI

4.2 Training Procedure and Results

The model was trained on the UCSD JupyterHub, which provided access to GPUs for fast training. The dataset was split with 75% used for training and 25% for validation. A compact, custom CNN was trained for 100 epochs using the Adam optimizer and cross-entropy loss. Data augmentation was applied in real time using `torchvision.transforms`, including brightness and contrast jitter, affine transformations, and translation shifts.

The final model achieved 91% validation accuracy. More importantly, it generalized well to real-time use cases, correctly identifying and reacting to obstacles like chair legs, backpacks, shoes, and walls. On live inference tests, the robot responded as expected, turning or reversing based on the visual input. These behaviors were validated in multiple trials by manually moving the robot to different positions and observing both printed predictions and physical motor output.

```python
import torch
import torchvision.transforms as transforms
from PIL import Image
import subprocess
import serial
import time
import os
from FinalCNN import FinalCNN
SERIAL_PORT = "/dev/ttyACM0"
BAUD_RATE = 9600
IMG_SIZE = 128
CAPTURE_PATH = "live.jpg"
CLASS_NAMES = ['clear', 'left', 'right', 'stop']
MODEL_PATH = "model.pt"

device = torch.device("cpu")
model = FinalCNN(num_classes=len(CLASS_NAMES))
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor()
])

try:
    arduino = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=1)
    time.sleep(2)
    print("Serial connected to Arduino.")
except Exception as e:
    print(f" connection failed: {e}")
    arduino = None

try:
    while True:
        subprocess.run([
            "libcamera-jpeg",
            "-o", CAPTURE_PATH,
            "--width", str(IMG_SIZE),
            "--height", str(IMG_SIZE),
            "--nopreview",
            "--timeout", "100"
        ], check=True)

        img = Image.open(CAPTURE_PATH).convert('RGB')
        input_tensor = transform(img).unsqueeze(0)

        with torch.no_grad():
            output = model(input_tensor)
            _, predicted = torch.max(output, 1)
            label = CLASS_NAMES[predicted.item()]

        print(f"Prediction: {label}")

        if arduino:
            arduino.write((label + "\n").encode())

except KeyboardInterrupt:
    print("Leaving")

if arduino:
    arduino.close()
```

Figure 10: Inference Module Run in Live time

4.3 Design Insights

During early experimentation, I included common image augmentations such as random rotation and horizontal flipping. However, these significantly hurt performance, especially for left/right classification, which relies on consistent spatial layout in the frame. After removing these spatially-destructive augmentations, accuracy rose from 54% to 91%, confirming the importance of directional consistency in camera-fixed systems.

Speed tuning was another critical part of the experimentation phase. Initially, the motors were too fast and unbalanced, especially during turning. Through iterative tests, I gradually reduced PWM values and ensured both wheels remained active during turns with one faster than the other to produce smoother arcs rather than jerky pivots. The reverse-then-spin behavior used for the `stop` class was carefully calibrated with timed delays, ensuring the robot reoriented predictably and re-entered exploration mode.

Additional small-scale experiments showed that even with reduced data with as few as 30–40 examples per class, the model could learn reasonably well, though performance dropped to the 70–80% range. This highlights the scalability of the system: with further data collection and more environmental diversity, this robot could be made robust enough to operate in new rooms, under varying lighting conditions, and around novel obstacle types.

Figure 11: Front View

Figure 12: Back View

4.4 Discussion

The experiment phase revealed how tightly coupled data collection, hardware control, and training logic must be for a visually driven robot. Every adjustment: from how the camera was mounted, to how the labeling script was written, to which obstacles were included had a tangible effect on downstream model behavior. This project successfully demonstrated that with careful dataset construction and thoughtful control logic, a purely vision-based robot can achieve meaningful, autonomous navigation in indoor environments without additional sensors.

5 Link to Video Demo

A video demonstration of the robot navigating an obstacle course is available here:
`Watch Project Demo Video`.

References

[1] Bojarski, Mariusz, et al. "End to end learning for self-driving cars." *arXiv preprint arXiv:1604.07316* (2016).

[2] Howard, Andrew G., et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

[3] OpenBot Project. `https://github.com/intel-isl/OpenBot`